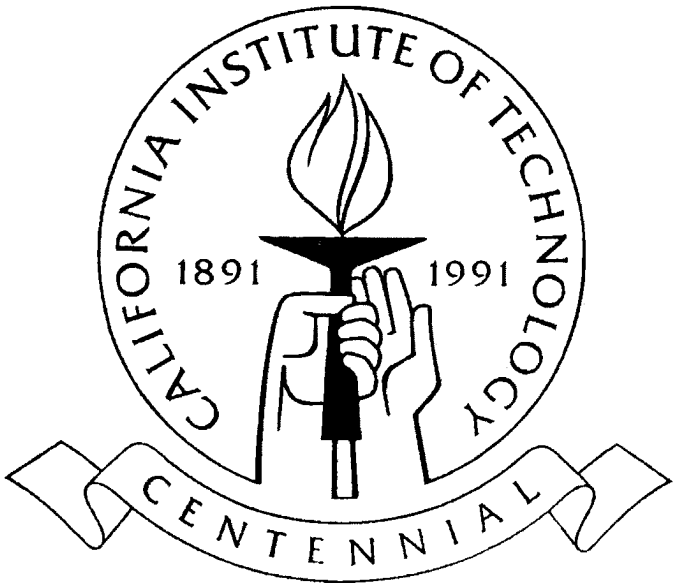

Using Triples to Reason About Concurrent Programs

K. Mani Chandy

Computer Science Department
California Institute of Technology

Caltech-CS-TR-93-02

Using Triples to Reason About Concurrent Programs

K. Mani Chandy
California Institute of Technology 256-80,
Pasadena, California 91125
mani@vlsi.caltech.edu *

January 20, 1993

Abstract

This paper presents adapatations of the Hoare triple for reasoning about concurrent programs. The rules for the Hoare triple, familiar to programmers from their experience with sequential programming, can be applied to develop proofs of concurrent programs as well. The basis for the adaptations of the Hoare triple is temporal logic.

1 Introduction

1.1 Goal

The introduction of the Hoare triple [4], more than two decades ago, provided a mathematical foundation for reasoning about sequential programs. Now, programmers are familiar with reasoning using the Hoare triple for sequential programs. This paper suggests that the familiar rules for reasoning with Hoare triples can be applied to develop proofs of concurrent programs.

This paper does not propose a new logic for concurrent programs. The foundation is provided by temporal logic. UNITY [1] or TLA [6] can be used to provide the framework for the operators and triples proposed in this paper. This paper has the restricted goal of demonstrating Hoare-triple reasoning for concurrent programs.

1.2 Triples

A relation $F(U, P, V)$ where P is a program, and U, V are predicates on states or computations, is defined to be *Hoare-triple-like* if and only if it satisfies the following five formulae.

*Supported in Part by AFOSR 91-0070.

Strengthening left side

$$(U' \Rightarrow U) \wedge F(U, P, V) \;\Rightarrow\; F(U', P, V) \tag{1}$$

Weakening right side

$$F(U, P, V) \wedge (V \Rightarrow V') \;\Rightarrow\; F(U, P, V') \tag{2}$$

Disjunction of both left and right sides

$$F(U, P, V) \wedge F(U', P, V') \Rightarrow F(U \vee U', P, V \vee V') \tag{3}$$

Conjunction of both left and right sides

$$F(U, P, V) \wedge F(U', P, V') \Rightarrow F(U \wedge U', P, V \wedge V') \tag{4}$$

The Hoare-triple formula for sequential composition is not included in the above set because we shall propose Hoare-triple-like formulae for *parallel* composition. In place of the formula for sequential composition, we use the following formula:

Transitivity

$$F(U, P, V) \wedge F(V, P, W) \Rightarrow F(U, P, W) \tag{5}$$

The reasoning about the correctness of many concurrent programs can be based on these familiar rules. Using a single small set of rules for reasoning about sequential programs, safety properties of concurrent programs, progress properties of concurrent programs, and parallel composition, can be simpler than using many different sets of rules.

1.3 Proposal

In this paper, we suggest two Hoare-triple-like relations:

1. $U \hookrightarrow V$ in P, read "U to always V in P", to reason about progress properties.

2. $[U]\ P\ [V]$, read "U environment P V" to reason about the parallel composition of program P with other programs.

We shall also use a Hoare-triple-like relation: U *co* V in P, suggested by Misra [7], to reason about safety properties. A goal of this paper is to suggest that these three Hoare-triple-like relations, and the five rules (1) - (5) appear adequate for reasoning about many concurrent programs.

2 Operational Model

2.1 Introduction

The operational model is, in essence, the operational model used in UNITY [1]. A program can be thought of as a fair, **do-od** loop:

do $g_0 \rightarrow a_0 [] \ldots [] g_n \rightarrow a_n$ **od**

where n is finite, and for all i, a_i is a deterministic terminating parallel assignment. The fairness requirement is: If g_i holds at some point in an infinite computation then there exists a later point in the computation at which a_i is executed or g_i does not hold. The set of guarded commands of the parallel composition of programs is the union of the sets of guarded commands of the components. This operational model is defined in this section.

2.2 Definitions

Program A program is a finite tuple of variables, and a finite set of events. The state of a program is given by the tuple of values of its variables; the set of program states is the set of all such tuples. For completeness, we define the set of states of a program with an empty tuple of variables as a set consisting of a single state S_{empty}.

Program counters are treated as variables. Sequential composition is represented by events that modify program counters (and possibly other variables).

Event An event A is a triple (W, E, F):

1. **Variables relevant to the event:** W is a nonempty subset of the variables of the program.

2. **Enabledness of the event:** E is a predicate on W. The event (W, E, F) is defined to be *enabled* if and only if E holds.

3. **New values of variables modified by event:** F is a function that maps from W to W. If event A is enabled in a state S, and $W = W_0$ in state S, then the occurrence of event (W, E, F) in state S takes the system to a state in which $W = F(W_0)$ and the values of all other variables remain unchanged.

In terms of the fair **do-od** loop, the event (W, E, F) is the guarded command:

$$E \rightarrow W := F(W)$$

State Transitions Each state transition is labeled with a single event. There exists a transition from a state S to a state S' labeled with event A if and only if event A is enabled in state S, and the occurence of event A in state S takes the system to state S'. Therefore, there exists a transition from S to S' labeled with an event A, where $A = (W, E, F)$, if and only if

1. E holds in state S, and
2. the values of all variables other than those in W are the same in S and S', and
3. if $W = W_0$ in state S, then $W = F(W_0)$ in state S'.

Since F is a function, for a given state S and a given event A, there exists at most one transition from S labeled A.

If there exists a transition from a state S to a state S' labeled A, then we use $A(S)$ to denote state S'. If there exists no transition from S labeled A then $A(S)$ is undefined.

There can be an arbitrary number of transitions (each labeled with a separate event) from a state S to a state S'.

Computation A computation is a state S_0 and a sequence of events A_i, $i > 0$, where the i-th state S_i reached in the computation is defined recursively as follows:

$$(\forall i : i > 0 : A_i \text{ is enabled in } S_{i-1} \ \wedge \ S_i = A_i(S_{i-1})) \tag{6}$$

and where the sequence obeys the fairness rule, given next.

Fairness The fairness rule is that in each infinite computation, if an event is enabled at some point in the computation, then there is a later point in the computation at which the event occurs or is disabled.

For all events B of the program, and all infinite computations with initial state S_0, and sequence of events $A_i, i > 0$, and for all j:

$(B \text{ is enabled in } S_j) \Rightarrow (\exists k : j < k : (A_k = B) \vee \neg(B \text{ is enabled in } S_k))$

where S_i, all i are defined in equation (6).

Maximal Computations A state S is defined to be a terminal state if and only if there are no events enabled in S. A maximal computation is defined to be an infinite computation or a computation ending in a terminal state.

Programs and Concurrent Composition The parallel composition of programs P and Q is a program denoted by $P||Q$. The tuple of program variables of $P||Q$ is the union of the tuples of program variables of P and Q. The set of events of $P||Q$ is the union of the sets of events of P and Q.

Since union is associative, commutative and idempotent, it follows that parallel composition is also associative, commutative and idempotent.

The identity element of parallel composition is a program with an empty tuple of variables and an empty set of events.

3 Program Properties

3.1 To Always

Let U and V be predicates on the states of a program P. We introduce a boolean, $U \hookrightarrow V$ in P, defined as follows:

for all maximal computations C of P: if U holds at any point i in C then there is a point j in C at which V holds, and after which V continues to hold.

$(U \hookrightarrow V)$ in P $=$
$(\forall$ maximal computations C of $P ::$
$\quad (\exists i : 0 \leq i \leq \mathit{length}(C) : U$ holds in $S_i) \Rightarrow$
$\quad\quad (\exists j : 0 \leq j \leq \mathit{length}(C) : (\forall k : j \leq k \leq \mathit{length}(C) : V$ holds in $S_k)))$

where S_i is defined in equation (6), and $\mathit{length}(C)$ is the number of events in C.

Theorem 1 *The triple $U \hookrightarrow V$ in P is Hoare-triple-like.*

Proof: For brevity, we shall use C for a maximal computation of P, and i, j, j' for integers where $0 \leq i, j, j' \leq \mathit{length}(C)$, and k for an integer where $j \leq k \leq \mathit{length}(C)$, and k' for an integer where $j' \leq k' \leq \mathit{length}(C)$.

Proof of strengthening left side:

$(U' \Rightarrow U) \wedge (U \hookrightarrow V$ in $P)$

$\Rightarrow$ $\quad \{ (U' \Rightarrow U) \Rightarrow (\forall i :: (U'$ holds in $S_i) \Rightarrow (U$ holds in $S_i)) \}$

$(\forall C :: (\exists i :: U'$ holds in $S_i) \Rightarrow (\exists i :: U$ holds in $S_i)) \wedge$
$(U \hookrightarrow V$ in $P)$

$=$ $\quad \{$ definition of $\hookrightarrow \}$

$(\forall C :: (\exists i :: U' \text{ holds in } S_i) \Rightarrow (\exists i :: U \text{ holds in } S_i)) \ \wedge$
$(\forall C :: (\exists i :: U \text{ holds in } S_i) \Rightarrow (\exists j, \forall k :: V \text{ holds in } S_k))$

$\Rightarrow$ {properties of $\forall$, and transitivity of $\Rightarrow$ }

$(\forall C :: (\exists i :: U' \text{ holds in } S_i) \Rightarrow (\exists j, \forall k :: V \text{ holds in } S_k))$

$=$ { definition of $\hookrightarrow$ }

$U' \hookrightarrow V$ in P

The proof of weakening the right side is very similar, and is left to the reader.

Proof of conjunction: Assume $(U \hookrightarrow V \text{ in } P) \wedge (U' \hookrightarrow V' \text{ in } P)$. The proof is carried out for any maximal computation C.

$(\exists i :: U \wedge U' \text{ holds in } S_i)$

$\Rightarrow$ { meaning of "holds in" }

$(\exists i :: (U \text{ holds in } S_i) \wedge (U' \text{ holds in } S_i))$

$\Rightarrow$ { property of $\exists$ }

$(\exists i :: (U \text{ holds in } S_i)) \wedge (\exists i :: (U' \text{ holds in } S_i))$

$\Rightarrow$ { $(U \hookrightarrow V \text{ in } P) \wedge (U' \hookrightarrow V \text{ in } P)$ }

$(\exists j, \forall k :: V \text{ holds in } S_k) \wedge (\exists j', \forall k' :: V' \text{ holds in } S_{k'})$

$=$ { for any j, j' satisfying this formula, set j to $\max(j, j')$ }

$(\exists j :: (\forall k :: V \text{ holds in } S_k) \wedge (\forall k :: V' \text{ holds in } S_k))$

$=$ {predicate calculus}

$(\exists j, \forall k :: (V \text{ holds in } S_k) \wedge (V' \text{ holds in } S_k))$

$=$ { meaning of "holds in" }

$(\exists j, \forall k :: V \wedge V' \text{ holds in } S_k)$

The proof follows.

Proof of disjunction: Assume $(U \hookrightarrow V \text{ in } P) \wedge (U' \hookrightarrow V' \text{ in } P)$. The proof is carried out for any maximal computation C.

$$
\begin{array}{ll}
 & (\exists i :: U \vee U' \text{ holds in } S_i) \\
= & \quad \{ \text{ meaning of "holds in" } \} \\
 & (\exists i :: (U \text{ holds in } S_i) \vee (U' \text{ holds in } S_i)) \\
= & \quad \{ \text{ property of } \exists \} \\
 & (\exists i :: U \text{ holds in } S_i) \vee (\exists i :: U' \text{ holds in } S_i) \\
\Rightarrow & \quad \{ (U \hookrightarrow V \text{ in } P) \wedge (U' \hookrightarrow V \text{ in } P) \} \\
 & (\exists j, \forall k :: V \text{ holds in } S_k) \vee (\exists j, \forall k :: V' \text{ holds in } S_k) \\
= & \quad \{ \text{ property of } \exists \} \\
 & (\exists j :: (\forall k :: V \text{ holds in } S_k) \vee (\forall k :: V' \text{ holds in } S_k)) \\
\Rightarrow & \quad \{ \text{ property of } \forall \} \\
 & (\exists j :: (\forall k :: (V \text{ holds in } S_k) \vee (V' \text{ holds in } S_k))) \\
= & \quad \{ \text{ meaning of "holds in" } \} \\
 & (\exists j :: (\forall k :: V \vee V' \text{ holds in } S_k))
\end{array}
$$

The proof follows.

The proof of transitivity is similar, and is left to the reader.

3.2 co

The definition of *co*, adapted from Misra [7], is

$$
\begin{array}{l}
(U \ co \ V \ \text{ in } P) = \\
\quad (U \Rightarrow V) \wedge (\forall \text{ events } (W, E, F) \text{ of } P :: \quad \{U \wedge E\}\ W := F(W)\ \{V\}\ )
\end{array}
$$

Misra has shown from the properties of Hoare triples, and transitivity of $\Rightarrow$, that $U \ co \ V$ in P is Hoare-triple-like, and since the set of events of $P \| Q$ is the union of the sets of events of P and Q:

$$(U \ co \ V \text{ in } P) \wedge (U \ co \ V \text{ in } Q) \ = \ (U \ co \ V \text{ in } P \| Q) \qquad (7)$$

Misra has also shown that (U *co* V in P) holds if and only if $U \Rightarrow V$, and for all state transitions in P from a state S to a state S', if U holds in state S then V holds in state S'.

3.3 Proving To-Always Properties

We can prove $U \hookrightarrow V$ in P using the UNITY rule, see [1]:

$((U \wedge \neg V)$ *co* $(U \vee V)$ in $P) \ \wedge \ (V$ *co* V in $P) \ \wedge$
$(\exists$ event (W, E, F) in $P :: ((U \wedge \neg V) \Rightarrow E) \ \wedge \ \{U \wedge \neg V\} \ W := F(W) \ \{V\})$
$\Rightarrow U \hookrightarrow V$ in P

The rule follows from:

1. $(U \wedge \neg V)$ *co* $(U \vee V)$ in P implies that if $U \wedge \neg V$ holds for a state S in a computation of P, and S is not a terminal state of P, then in the next state either $U \wedge \neg V$ continues to hold, or V holds; therefore $U \wedge \neg V$ continues to hold forever, or eventually V holds [1].

2. $(\exists$ event (W, E, F) in $P :: ((U \wedge \neg V) \Rightarrow E) \wedge \{U \wedge \neg V\} \ (W, E, F) \ \{V\})$ implies that $U \wedge \neg V$ cannot continue to hold forever, because if $U \wedge \neg V$ holds then the event (W, E, F) is enabled, and from the fairness rule, eventually $W := F(W)$ will be executed, and that makes V hold.

3. V *co* V in P implies that if V holds at any point in a computation of P then it continues to hold forever thereafter.

We can also use the five Hoare-triple rules in proving $U \hookrightarrow V$ in P. A particularly valuable rule is weakening the right side. In sequential programming, to prove that a predicate $\mathcal{I}$ always holds, we may have to prove that a stronger predicate than $\mathcal{I}$ is an invariant. Likewise, to prove $U \hookrightarrow V$ in P we may have to prove $U \hookrightarrow W$ in P where W is stronger than V.

Theorem 2

$$(U \text{ co } U \text{ in } P) \ \Rightarrow \ (U \hookrightarrow U \text{ in } P)$$

Proof: Follows from the proof rule for $\hookrightarrow$, substituting U for V.

4 Compositional Triples

Compositional triples are used in proving parallel composition of programs, given the specifications (but not the program texts) of the components.

Program Properties A program-property is defined to be a predicate on programs. We use the abbreviation "property" for program-property where no ambiguity results.

For a property α and a program P, we use the notation

$$\alpha \text{ in } P$$

to denote the boolean: property α holds for program P. We define:

$$\begin{aligned} (\alpha \wedge \beta) \text{ in } P &= (\alpha \text{ in } P) \wedge (\beta \text{ in } P) \\ (\neg\alpha) \text{ in } P &= \neg(\alpha \text{ in } P) \end{aligned} \tag{8}$$

Hence,

$$(\alpha \vee \beta) \text{ in } P = (\alpha \text{ in } P) \vee (\beta \text{ in } P)$$

Examples of properties that we use in reasoning about concurrent programs are U *co* V, and $U \hookrightarrow V$.

Let U and V be properties, and let P be a program. A compositional triple is a boolean, defined as follows:

$$[U]\ P\ [V] = (\forall \text{ programs } Q : U \text{ in } Q||P : V \text{ in } Q||P)$$

The triple says that if we restrict attention to environments of P such that U is a property of P composed with its environment, then V is a property of P composed with its environment.

Theorem 3 *The relation* $[U]\ P\ [V]$ *is Hoare-triple-like.*

Proof:

Proof of strengthening left side

$(U' \Rightarrow U) \wedge ([U]P[V])$

$\Rightarrow$ { $(U' \Rightarrow U) \Rightarrow (\forall Q : U' \text{ in } Q||P : U \text{ in } Q||P)$ }

$(\forall Q : U' \text{ in } Q||P : U \text{ in } Q||P) \wedge ([U]P[V])$

$=$ { definition of $[U]P[V]$ }

$(\forall Q : U' \text{ in } Q||P : U \text{ in } Q||P) \wedge (\forall Q : U \text{ in } Q||P : V \text{ in } Q||P)$

$\Rightarrow$ { predicate calculus }

$(\forall Q : U' \text{ in } Q||P : V \text{ in } Q||P)$

$=$ { definition of $[U']P[V]$ }

$[U']P[V]$

The proof of weakening the right side is similar.

Proof of conjunction

$([U]P[V]) \wedge ([U']P[V'])$

$\Rightarrow$ { definition of compositional triple }

$(\forall Q : U \text{ in } Q\|P : V \text{ in } Q\|P) \wedge (\forall Q : U' \text{ in } Q\|P : V' \text{ in } Q\|P)$

$=$ { predicate calculus }

$(\forall Q : (U \text{ in } Q\|P) \wedge (U' \text{ in } Q\|P) : (V \text{ in } Q\|P) \wedge (V' \text{ in } Q\|P))$

$=$ { equation (8) }

$(\forall Q : (U \wedge U' \text{ in } Q\|P) : (V \wedge V' \text{ in } Q\|P))$

$=$ { definition of compositional triple }

$[U \wedge U']P[V \wedge V']$

Proofs of disjunction and transitivity are similar.

Proofs of Composition The following two theorems are helpful in compositional design of concurrent programs.

Theorem 4 *Inheritance: A parallel block inherits compositional triples of its components.*

$$([U]\ Q\ [V]) \ = \ (\forall Q' :: ([U]\ Q\|Q'[V]) \tag{9}$$

Proof:

$[U]\ Q\ [V]$

$=$ {definition of compositional triple}

$(\forall P : U \text{ in } Q\|P : V \text{ in } Q\|P)$

$\Rightarrow$ {setting P to $Q'\|P'$ }

$(\forall Q', P' : U \text{ in } Q\|Q'\|P' : V \text{ in } Q\|Q'\|P')$

$=$ { predicate calculus }

$$(\forall Q' :: (\forall P' :: U \text{ in } Q\|Q'\|P' : V \text{ in } Q\|Q'\|P'))$$

$=$ {definition of compositional triple}

$$(\forall Q' :: [U]\ Q\|Q'\ [V])$$

The proof that

$$(\forall Q' :: ([U]\ Q\|Q'[V])) \Rightarrow ([U]\ Q\ [V])$$

follows by setting Q' to Q, since $Q\|Q = Q$.

Corollary For any given set of programs $P_0 \ldots P_n$:

$$(\forall i : 0 \leq i \leq n : [U_i]\ P_i\ [V_i]) \Rightarrow (\forall i : 0 \leq i \leq n : [U_i]\ \ P_0\| \ldots \|P_n\ \ [V_i])$$

Proof: Follows from the last theorem, and the associativity and commutativity of parallel composition.

Theorem 5 *The derivation of a compositional triple for a parallel composition from the compositional triples of its components.*

For any given set of programs $P_0 \ldots P_n$*:*

$$(\forall i : 0 \leq i \leq n : [U_i]\ P_i\ [V_i]) \Rightarrow$$
$$[(\forall i : 0 \leq i \leq n : U_i)]\ \ P_0\| \ldots \|P_n\ \ [(\forall i : 0 \leq i \leq n : V_i)]$$

Proof: Follows from the previous corollary and conjunctivity.

Example An example of a compositional triple is given next.

Let U and V be predicates on states of programs. Let (W, E, F) be an event of a program P such that

$$((U \wedge \neg V) \Rightarrow E)\ \wedge\ \{U \wedge \neg V\}\ W := F(W)\ \{V\}$$

Define a program property Z as follows:

$$Z\ \ =\ \ ((U \wedge \neg V)\ co\ (U \vee V))\ \wedge\ (V\ co\ V)$$

Then:

$$[Z]P[U \hookrightarrow V]$$

Proof: For all programs Q:

Z in $P \| Q$

$\Rightarrow$ { definition of Z, and existence of event (W, E, F) in P }

$((U \wedge \neg V)\ co\ (U \vee V)$ in $P \| Q) \wedge (V\ co\ V$ in $P \| Q) \wedge$
$(\exists$ event (W, E, F) in $P \| Q :: ((U \wedge \neg V) \Rightarrow E) \wedge \{U \wedge \neg V\}\ W := F(W)\ \{V\})$

$\Rightarrow$ {proof rule for $\hookrightarrow$}

$U \hookrightarrow V$ in $P \| Q$

5 Example

5.1 Proof Restrictions

In our proofs we place two significant restrictions on ourselves:

1. The only proof rules we use are the Hoare-triple rules, and the basic rules for proving $\hookrightarrow$ and *co*.
2. Our proofs are compositional: the specification of a composition of programs is proved from specifications *but not the program texts* of the components.

A potential concern is that by restricting ourselves in this way we we will produce proofs that are much longer than if we use the full power of temporal logic or if we use program texts. Our hypothesis is that many programs can be designed so that they can be proved with the restrictions, without significantly increasing the length or complexity of proofs.

In the next section we present a simple example of a concurrent program proof using triples.

5.2 The Problem

The problem is a slightly more complex version of the *earliest meeting time* example in [1]. A parallel composition of three professors, P_i, $0 \leq i < 3$, and a secretary Sec, computes the earliest time at which all three professors can meet. Time ranges over the nonnegative integers.

Professor P_i has input x and output y_i, all i. Associated with P_i is a function f_i, where $f_i(k)$ is the earliest time at or after k that P_i can meet. If $y_i < f_i(x)$ then P_i nondeterministically selects some value in the range $y_i \ldots f_i(x)$, and sets y_i to this value. If y_i remains less than $f_i(x)$ then eventually P_i will increase y_i.

The secretary Sec has input y_i, all i, and output x. If $x < \max_i y_i$, then Sec nondeterministically selects some value in the range $x \ldots \max_i y_i$, and sets x to this value. If x remains less than $\max_i y_i$ then, eventually Sec will increase x.

Let e be the earliest common meeting time of all professors:

$$e = (\min t : t \geq 0 \wedge t = f_i(t) : t)$$

We are given that e exists and is finite.

We wish to prove that if $(0 \leq x \leq e) \wedge (\forall i :: y_i \leq e)$ at any point in a computation, then eventually $(x = e) \wedge (\forall i :: y_i = e)$ in the computation.

A sequential program that computes e is:

$x := 0$; **while** $x \neq g(x)$ **do** $x := g(x)$; $\quad$ { Assert: $x = e$ }

where $g(x) = \max_i f_i(x)$. An invariant of the loop is $x \leq e$, since $0 \leq e$, and

$$x \leq e \;\Rightarrow\; g(x) \leq e$$

A variant function for the loop is $e - x$. At termination of the program, $x = g(x)$, and the loop invariant holds, and hence $x = e$.

From the program, it follows that there exists a finite N such that

$$g^N(0) = e$$

where g^k is k successive applications of function g.

Next, we give the formal specification of the problem.

5.3 Specification of Professors

1. Variables of P_i are x and y_i.

2. Professor i does not modify x:

$$(x = k) \; co \; (x = k) \text{ in } P_i$$

3. Professor i does not decrease y_i:

$$\mathit{nondecreasing}(y_i) \text{ in } P_i$$

where the property $\mathit{nondecreasing}(z)$ is defined as

$$\mathit{nondecreasing}(z) \;=\; (\forall k :: (z \geq k) \; co \; (z \geq k))$$

4. Professor i does not set y_i to larger than the next meeting time at or after x.

$$(x \leq k) \wedge (y_i \leq f_i(k)) \;\; co \; (y_i \leq f_i(k)) \;\; \text{in } P_i$$

5. If x and y_i are nondecreasing in any program that has P_i as one of its components, then $(x \geq k) \hookrightarrow (y_i \geq f_i(k))$ in that program.

$$[\mathit{nondecreasing}(x, y_i)] \;\; P_i \;\; [(x \geq k) \hookrightarrow (y_i \geq f_i(k))]$$

where the property $\mathit{nondecreasing}(x, y_i)$ is defined as:

$$\mathit{nondecreasing}(x, y_i) = \mathit{nondecreasing}(x) \wedge \mathit{nondecreasing}(y_i)$$

5.4 Specification of the Secretary

1. Variables of *Sec* are x, and $y_i, i = 0, 1, 2$.

2. The secretary does not modify y_i, all i.

$$(\forall i :: y_i = m_i) \; co \; (\forall i :: y_i = m_i) \text{ in } Sec$$

3. The secretary does not decrease x.

$$nondecreasing(x) \text{ in } Sec$$

4. The secretary does not set x to larger than the maximum of the y_i all i.

$$(\forall i :: y_i \leq m_i) \wedge (x \leq \max_i m_i) \;\; co \; (x \leq \max_i m_i) \;\; \text{in } Sec$$

5. If x and y_i, all i, are nondecreasing in any program that has *Sec* as one of its components, then $(\forall i :: y_i \geq m_i) \hookrightarrow (x \geq \max_i m_i)$ in that program.

$$[nondecreasing(x, y_0, y_1, y_2)] \;\; Sec \;\; [(\forall i :: y_i \geq m_i) \hookrightarrow (x \geq \max_i m_i)]$$

5.5 Stability

Let:

$$W \;\; = \;\; (x \leq e) \wedge (\forall i :: y_i \leq e)$$

Let us prove

$$W \; co \; W \text{ in } P_i$$

Proof:

From the specification of P_i:

$$(x \leq k) \wedge (y_i \leq f_i(k)) \;\; co \; (y_i \leq f_i(k)) \;\; \text{in } P_i$$

$\Rightarrow$ {substituting e for k, and using $f_i(e) = e$}

$$(x \leq e) \wedge (y_i \leq e)) \;\; co \; (y_i \leq e) \;\; \text{in } P_i$$

$\Rightarrow$ {Using conjunctivity, and $(y_j \leq e) \;\; co \; (y_j \leq e) \;\; \text{in } P_i$, for $j \neq i$ since y_j is not a variable of P_i, and $(x \leq e) \; co \; (x \leq e)$ in P, since x is not modified by P_i}

$$W \; co \; W \text{ in } P_i$$

The proof of $W \; co \; W$ in *Sec* is very similar and is not given here. The proof of $W \; co \; W$ in R, where $R = P_0 || P_1 || P2 || Sec$ follows from the compositionality of *co*, equation (7).

Monotonicity Next we prove:

$$(x \geq k)\ co\ (x \geq k) \text{ in } R$$

Proof: From the specification of Sec:

$$(x \geq k)\ co\ (x \geq k) \text{ in } Sec$$

Since x is not modified by P_i, all i.

$$(x \geq k)\ co\ (x \geq k) \text{ in } P_i$$

From the above two equations and equation (7)

$$(x \geq k)\ co\ (x \geq k) \text{ in } R$$

The proof of $(y_i \geq m_i)\ co\ (y_i \geq m_i)$ in R is almost identical.

5.6 Progress

Next we shall prove:

$$(x \geq 0) \hookrightarrow (x \geq e) \text{ in } R$$

Proof:

From the specification of P_i:

$$(\ \forall i :: [nondecreasing(x, y_i)]\ P_i\ [x \geq k \hookrightarrow y_i \geq f_i(k)]\ )$$

$\Rightarrow$ { we have proved $nondecreasing(x, y_i)$ all i in R }

$$(\forall i :: x \geq k \hookrightarrow y_i \geq f_i(k) \text{ in } R)$$

$\Rightarrow$ {Conjunction}

$$(x \geq k) \hookrightarrow (\forall i :: y_i \geq f_i(k)) \text{ in} R$$

$\Rightarrow$ { We can prove similarly, using the specification of Sec
$(\forall i :: y_i \geq f_i(k)) \hookrightarrow (x \geq \max_i f_i(k))$ in R
and using transitivity, and $g(k) = \max_i f_i(k)$}

$$(x \geq k) \hookrightarrow (x \geq g(k)) \text{ in } R$$

$\Rightarrow$ {Transitivity}

$$(x \geq k) \hookrightarrow (x \geq g^N(k)) \text{ in } R$$

$\Rightarrow$ {Substitute 0 for k, and use $g^N(0) = e$}

$$(x \geq 0) \hookrightarrow (x \geq e) \text{ in } R$$

5.7 Termination

Finally we prove:

$$(0 \leq x \leq e) \wedge (\forall i :: y_i \leq e) \;\hookrightarrow\; (x = e) \wedge (\forall i :: y_i = e) \text{ in } R$$

Proof:

We have shown:

$$(x \geq 0) \hookrightarrow (x \geq e) \text{ in } R$$

$\Rightarrow$ {transitivity, using $x \geq k \;\hookrightarrow\; y_i \geq f_i(k)$ in R, which we proved in the last theorem, and substituting e for k, and using $f_i(e) = e$, and taking conjunction over all i }

$$(x \geq 0) \hookrightarrow (\forall i :: y_i \geq e) \text{ in } R$$

$\Rightarrow$ {conjunction of the last two and $W \hookrightarrow W$}

$$(0 \leq x \leq e) \wedge (\forall i :: y_i \leq e) \;\hookrightarrow\; (x = e) \wedge (\forall i :: y_i = e) \text{ in } R$$

6 Evaluation, Further Work, and Past Work

6.1 Monotonicity, Auxiliary Variables and Progress

Part of the specification for a mutual-exclusion program has the form: If P is waiting to enter its critical section then in a finite number of steps, P will be in its critical section. The predicate *in critical section* holds for only a finite number of steps, so

$$\text{waiting for critical section} \hookrightarrow \text{in critical section}$$

does not hold for a mutual-exclusion program.

Often, the progress property for mutual exclusion is specified as:

$$\text{waiting for critical section} \rightsquigarrow \text{in critical section}$$

where $U \rightsquigarrow V$ is defined as: if U holds at some point in a computation then V holds at a later point in the computation [6, 1]. But, $U \rightsquigarrow V$ in P is not Hoare-triple-like because it does not satisfy conjunctivity. A central question for the approach proposed in this paper is: *Can we specify and reason about concurrent programs using only triples that satisfy the five rules: (1) - (5) ?* This issue is explored in the next two paragraphs.

Programs are designed with some concept of "progress" in computations — informally speaking, more has been accomplished at a later point in a computation than at an earlier point in the computation. The concept of

progress in terms of "more being accomplished" is appropriate even for non-terminating programs such as database systems: for instance, more transactions have been processed later in the computation. The notion of progress can be captured by variables (or auxiliary variables) that are monotone non-decreasing, and where as more gets accomplished, the variable gets larger. The value of such a variable is a measure (and there can be many measures) of progress, and so we call such variables *progress variables*. In a database system, an example of a progress variable is the number of transactions that have been processed.

In the case of the mutual-exclusion problem, examples of progress variables are $p.nw$, the number of times that a process p transits from not-waiting to waiting to enter critical section, and $p.ne$, the number of times that process p enters its critical section. We can specify the progress property in terms of progress variables:

$$(p.nw \geq k) \quad \hookrightarrow \quad (p.ne \geq k)$$

because "p has entered its critical section at least k times" is a stable property for any k.

Many problems, including those in [1] can be specified using $\hookrightarrow$ rather than $\rightsquigarrow$. What, then, are the disadvantages of $\hookrightarrow$?

The primary disadvantage appears to be that the use of $\hookrightarrow$ often requires the introduction of progress variables, and the introduction of auxiliary variables can lead to over-specification, because auxiliary variables can be defined in terms of an implementation. The introduction of $p.nw$ and $p.ne$ (and similar progress variables for other problems) does not, however, appear to result in over-specification.

6.2 Compositional Triples

Safety properties are compositional, equation(7). Progress properties are not, in general, compositional:

$$(U \hookrightarrow V \text{ in } P) \wedge (U \hookrightarrow V \text{ in } Q) \quad \not\Rightarrow (U \hookrightarrow V \text{ in } P||Q)$$

Compositional triples provide a way for proving specifications of parallel compositions of programs from specifications, but not the program texts, of components. Usually, in a compositional triple, the right side is a progress property, and the left side is a safety property or a conjunction of safety and progress properties.

We could have defined compositional triples in the following way: if the environment Q of P has a property U, then $P||Q$ has property V. This definition was rejected because it does not yield the theorem on inheritance of compositional triples, a central theorem in our approach to the design of concurrent programs. Also, this definition does not yield transitivity.

6.3 Future Work

Weakest Environments An intriguing line of research is to explore *weakest environment*, an analog to weakest precondition [2] for compositional triples. For a given program P and property V can we compute a weakest environment – a weakest property U such that $[U]P[V]$ holds? If it can be computed, how can it be used in program derivation?

For example, consider a program P with a single variable x, and a single event that is always enabled and increments x by 1. The program can be represented by the **do** - **od** loop;

$$\textbf{do } \mathit{true} \ \rightarrow \ x := x + 1 \textbf{ od}$$

Let V be the property:

$$(\forall j, k :: (x \geq j) \hookrightarrow (x \geq j + k))$$

The weakest U that satisfies $[U]P[V]$ is

$$\mathit{nondecreasing}(x)$$

Is it possible to derive a calculus of concurrent program derivation based on weakest environments? Is there a similar calculus for progress properties using $\hookrightarrow$?

Mechanical Proof Checkers Since the proof method proposed in this paper is based on Hoare-triple rules, mechanical proof checkers for sequential programs based on Hoare triples should be extensible to handle proofs of concurrent programs using our proof method. How difficult is such an extension?

Completeness We need to explore the completeness of the rules for $\hookrightarrow$. The rule may be incomplete even with the use of auxiliary variables.

Proper Concurrent Composition To what extent can proofs of concurrent programs be simplified by restricting parallel composition? For instance, let us define the parallel composition of programs P and Q to be *proper* if and only if in $P||Q$, a variable can be modified by at most one program — either P or Q but not both. In this case:

$$U \ co \ V \text{ in } P \quad \Rightarrow \quad U \ co \ V \text{ in } P||Q$$

if V references only variables modified by P. Likewise, proofs of progress properties can also be simplified.

Many concurrent programs, especially programs using message-passing, are structured so that parallel composition is proper. A line of research is to derive more powerful rules for proper parallel composition.

Evaluating the Hypothesis Our hypothesis is that most concurrent programs can be proved using only the well-known simple rules for Hoare triples; more complex rules and logics are not needed. This hypothesis needs to be evaluated by studying a large class of examples.

6.4 Past Work

This paper is based on UNITY [1]. The earliest proof methods for concurrent programming used Hoare triples, and were proposed in Owicki and Gries [8]. The proof method proposed in this paper is different from that proposed by Owicki and Gries in that noninterference in this paper is captured by compositional triples which are designed to help in proving concurrent programs without using the texts of components.

Compositional triples are very similar in spirit to the rely-guarantee approach proposed by Cliff Jones [5]. There are, however, differences in the definitions. Compositional triples deal with parallel composition of any environment and a program P, such that the composed program has property U — thus the *rely* property is a property of P and its environment.

This paper is motivated by Hoare's work on Hoare triples [4]. The proof structure in this paper is from Dijkstra and Scholten [3]

7 Acknowledgment

Thanks at Caltech to U. Binau, M. van der Goot, P. Hofstee, R. Leino, B. Massingill, A. Rifkin, P. Sivilotti, J. van de Snepscheut, J. Thornley, and J. Tierno, for their suggestions. Thanks to E. Knapp, L. Lamport, J. Misra and A. Singh for their careful reading. Special thanks to Rustan Leino for pointing out the identity element and the idempotence of parallel composition, and for strengthening some theorems by replacing implication by equality.

References

[1] Chandy, K. M. and J. Misra *Parallel Program Design: A Foundation*, Addison-Wesley, Reading, Massachusetts, 1988.

[2] Dijkstra, E.W., *A Discpline of Programming*, Prentice-Hall, Englewood Cliffs, N.J. 1976.

[3] Dijkstra, E.W., and C.S. Scholten, *Predicate Calculus and Program Semantics*, Springer-Verlag, New York, 1990.

[4] Hoare, C. A. R., "An Axiomatic Basis for Computer Programming," *CACM*, Vol. 12, 1969, pp 576-580.

[5] Jones, C.B., *Systematic Software Development Using VDM*, Prentice-Hall International, Englewood Cliffs, N.J. 1986.

[6] Lamport, L., "A Temporal Logic of Actions," Digital Equipment Corp. Systems Research Center, Palo Alto, tech report 57, April 1990.

[7] Misra, J., "Safety Properties" Report from Computer Sciences Dept., Univ. of Texas, Austin, TX78712, July 24, 1992.

[8] Owicki, S., and D. Gries, "An Axiomatic Proof Technique for Parallel Programs I," *Acta Informatica*, vol. 6, no.1, 1976, pp 319-340.